

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Utility Patent Application

REVIEWING THE SECURITY OF TRUSTED COMPONENTS

Inventor(s):

Cédric Fournet

Andrew Gordon

Tomasz Blanc

EL996276239

CLIENT'S DOCKET NO. MS306001.1

ATTORNEY'S DOCKET NO. MS1-1700US

REVIEWING THE SECURITY OF TRUSTED SOFTWARE COMPONENTS

Technical Field

The invention relates generally to execution environments, and a security model in an execution environment.

Background

It is a common scenario in networked computer systems for untrusted (or only partially trusted) program code to be received over the network (e.g., the Internet) and executed on a user's computer. Such untrusted code may include without limitation application updates, applets, plug-ins, user macros in documents, command-line scripts, programs downloaded from the network, etc. For example, a user may download a stock ticker applet written in a platform-independent intermediate language to be executed within a managed execution environment on the user's computer.

However, because the origin of the downloaded applet is unknown or untrusted, the user may not be confident in the safety of executing the applet on his or her computer, and rightly so – the applet code may include malicious instructions intended to steal or destroy the user’s data. Alternatively, while the code may not be malicious, improperly implemented code may inadvertently access or destroy the user’s data. Accordingly, such managed execution environments commonly embed a security framework to protect the user’s system.

Such security frameworks typically include a layered permission-based infrastructure. Varying permissions are attributed to each piece of code to be executed within the managed execution environment. Only code having the

1 necessary permissions are allowed to execute various operations (e.g., an
2 operation that accesses a system resource).

3 Also, as part of the layered aspect of the security framework, untrusted
4 code is isolated from sensitive system functions by more trusted components. In
5 one circumstance, an exemplary untrusted applet cannot directly access the file
6 system to modify a file. Instead, to modify a file in the file system, the untrusted
7 applet must call a trusted component (e.g., an object in a file system library
8 provided by the operating system vendor), which can access the file system.

9 However, the trusted component is also designed to adhere strictly to the
10 constraints of the security framework. Therefore, unless the untrusted applet is
11 attributed with adequate permissions to access the file system, and specifically to
12 modify the file, the trusted component should deny the requested access.
13 Moreover, the permissions required to access the file system may vary depending
14 on the type of access requested. For example, weak permissions may be required
15 to access a cookie file in a directory of cookies, while strong permissions may be
16 required to delete private user data elsewhere in the file system.

17 A significant strength of this security framework lies within the strict
18 adherence of the trusted components to the constraints of the security framework.
19 However, for a variety of reasons, even the security of highly trusted components
20 may be compromised in the presence of arbitrary untrusted code or improperly
21 implemented trusted code (e.g., when the highly trusted code does not actually
22 adhere strictly to the constraints of the security framework).

Summary

2 Implementations described and claimed herein address the foregoing
3 problems by providing an analysis tool for reviewing the security of trusted
4 software components during development. By examining the usage of
5 permissions in programs and libraries within the managed execution environment,
6 potential vulnerabilities in the security of trusted components may be identified.
7 In a first stage, a call graph generator creates a permission-sensitive call graph. In
8 a second stage, a call graph analyzer evaluates the permission-sensitive call graph
9 to highlight call paths that may present security risks. A developer can evaluate a
10 call paths set that includes some or all of the highlighted call paths to investigate
11 possible security vulnerabilities.

12 In some implementations, articles of manufacture are provided as computer
13 program products. One implementation of a computer program product provides a
14 computer program storage medium readable by a computer system and encoding a
15 computer program that generates a call graph. Another implementation of a
16 computer program product may be provided in a computer data signal embodied in
17 a carrier wave by a computing system and encoding the computer program that
18 generates a call graph.

19 The computer program product encodes a computer program for executing
20 on a computer system a computer process for generating a call graph. Input
21 component code and a runtime security policy are received into an execution
22 environment. A call graph is generated that includes call paths through the input
23 component code simulated in combination with at least one symbolic component
24 representing additional arbitrary code that complies with the runtime security
25 policy.

1 In another implementation, a method is provided. Input component code
2 and a runtime security policy are received into an execution environment. A call
3 graph is generated that includes call paths through the input component code
4 simulated in combination with at least one symbolic component representing
5 additional arbitrary code that complies with the runtime security policy.

6 In yet another implementation, a system is provided. A call graph
7 generator receives into an execution environment input component code and a
8 runtime security policy. The call graph generator generates a call graph of call
9 paths through the input component code simulated in combination with at least
10 one symbolic component representing additional arbitrary code that complies with
11 the runtime security policy.

12 Another implementation of a computer program product provides a
13 computer program storage medium readable by a computer system and encoding a
14 computer program that analyzes a call graph. Another implementation of a
15 computer program product may be provided in a computer data signal embodied in
16 a carrier wave by a computing system and encoding the computer program that
17 analyzes a call graph.

18 The computer program product encodes a computer program for executing
19 on a computer system a computer process for analyzing a call graph. A call graph
20 that includes call paths through input component code is analyzed. The call paths
21 are simulated in combination with at least one symbolic component that represents
22 additional arbitrary code that complies with a runtime security policy. A subset of
23 the call paths in the call graph that satisfy the query is identified.

24 In another implementation, a method is provided that includes analyzing a
25 call graph that includes call paths through input component code. The call paths

1 are simulated in combination with at least one symbolic component that represents
2 additional arbitrary code that complies with a runtime security policy. A subset of
3 the call paths in the call graph that satisfy the query is identified.

4 In another implementation, a system includes a call graph analyzer that
5 analyzes a call graph of call paths through input component code simulated in
6 combination with at least one symbolic component that represents additional
7 arbitrary code that complies with a runtime security policy. The analyzer analyzes
8 the call graph relative to at least one query and identifies a subset of the call paths
9 in the call graph that satisfy the query.

10 Other implementations are also described and recited herein.

11 **Brief Descriptions of the Drawings**

12 FIG. 1 illustrates an exemplary managed execution environment.

13 FIG. 2 illustrates an exemplary analysis tool for execution in a managed
14 execution environment.

15 FIG. 3 illustrates exemplary operations for generating a permission-
16 sensitive call graph.

17 FIG. 4 illustrates a system useful for implementing an embodiment of the
18 present invention.

20 **Detailed Description**

21 A feature of a managed execution environment described herein is that it
22 facilitates the deployment and integration of software components. For example,
23 the same component code may be used on different kinds of systems, one running
24 Windows and another running a variety of UNIX. Units of code deployment are
25

1 called “assemblies” and include portable code and metadata describing the code’s
2 intended usage and requirements. As pertains to security, the metadata, such as a
3 digital signature, may provide evidence of origin. At runtime, various assemblies
4 may also share the system resources, such as the same memory, the same stack,
5 the same namespace, and the same libraries.

6 Some operations of a program in a managed execution environment are
7 both useful and dangerous (e.g., send an email, delete a file, perform a system call,
8 etc.). Understandably, such operations should be enabled for some code, but not
9 all code. Therefore, an exemplary security framework keeps track of “code
10 identity”, i.e., the identity of the code that is responsible, either directly or
11 indirectly, for requesting the execution of the operation. Permissions are attached
12 to the identified code so that the operations that are available to the code are
13 limited by these permissions (i.e., limited to those operations allowed by the
14 permissions).

15 In one implementation, two complementary mechanisms are enforced:

- 16 (1) The rights attached to every piece of code are made explicitly
17 (e.g., according to the origin of the code and evidence attached
18 to the code). This assignment of rights to code and various
19 security checks performed as the code is loaded are referred to
20 as the “runtime security policy”.
- 21 (2) Before any security sensitive operation is executed, the rights of
22 any pieces of code responsible for the operation are examined.
23 This examination is referred to as “permission and stack
24 inspection”.

1 An exemplary stack inspection measure can determine the runtime rights of
2 each piece of code as a function of the call stack. Rights for a given piece of code
3 may be represented by static permissions attributed to the piece of code in
4 accordance with the code's level of trust. Then, before accessing a sensitive
5 resource, the call stack is inspected to verify that every caller in the call stack has
6 been granted the necessary rights.

7 Stack inspection is a strong security measure, but it presents complications
8 for programmers who are trying to verify that their code is secure. The behavior
9 of this code and, therefore, its security are strongly dependent upon the local
10 security framework and the runtime stack. In one implementation, an analysis tool
11 checks the use of permissions in trusted component code (e.g., trusted library
12 code) in an open system, where only some of the program code in the managed
13 execution environment is available at analysis-time. Based on this permission
14 check, potentially non-secure call paths are identified to allow the developer to
15 evaluate these possible security gaps. Some of the identified call paths may be
16 determined by the developer not to present a real security risk, while other
17 identified call paths may represent actual security problems with the trusted code.
18 Non-secure paths may be corrected by the developer when identified.

19 FIG. 1 illustrates a system 100 executing an exemplary managed execution
20 environment 102 as an application (e.g., a Common Language Runtime or CLR
21 application). An operating system 104 executes within the system 100 to manage
22 system resources and basic system functionality. The operating system 104
23 provides coarse access control to the system resources using security tokens and
24 access control lists (ACLs). Applications 106 and 102 access the operating
25 system 104 through controlled interface 108 (e.g., the Win32-x86 interface).

1 The application 106 is an example of typical application that is installed on
2 a user's computer by the user or an administrator. The application 106 may be
3 executed directly from the operating system 104 and generally has access to
4 system resources directly through the operating system 104.

5 The application 102 is an example of a special application that manages
6 execution of portable programs written in any of several supported languages,
7 allowing them to share common object-oriented classes written in any of the
8 supported languages as well as available system resources (e.g., CPU, network,
9 files, etc.). These portable programs may have diverse and even unknown origins
10 and they may include application updates, applets, plug-ins, user macros in
11 documents, command-line scripts, programs downloaded from the network, etc.
12 Portable programs generally present a higher-level of security risk than the
13 application 106. Therefore, the application 102 manages the execution of such
14 portable programs and performs fine-grained access control using features such as
15 types, permissions, stack inspections, etc. to protect the system resources.
16 Furthermore, these programs may share system resources, but they may do so with
17 varying levels of trust. Therefore, some portable programs may have more access
18 than others within the security framework of the managed execution
19 environment 102.

20 A managed applet 110 is shown as an example of a portable program that
21 executes within the managed execution environment. The application 102
22 executes the applet 110 and manages its access to system resources in accordance
23 with the applet's permissions and the existing security framework. The applet's
24 level of trust is represented by the static permissions that are attributed to it.

1 One aspect of the security framework involves a set of one or more trusted
2 components 112 executing within the managed execution environment. The
3 trusted components 112 may be provided or certified by the operating system
4 vendor, the managed execution environment vendor, or some other trusted vendor.
5 The trusted components 112 generally have a higher level of trust than the
6 applet 110. The higher level of trust enjoyed by these trusted components 112
7 allows them to have more static permissions and a greater level of access to the
8 system resources. As such, the applet 110 typically accesses the trusted
9 components 112 through a controlled interface 114 and accesses the system
10 resources indirectly through the trusted components 112. It should be understood
11 that there may be many levels of components, all of potentially varying levels of
12 trust, executing within the managed execution environment and executing in
13 cooperation with the applet 110.

14 The trusted component code typically includes security demands to the
15 security framework, which determines whether a piece of sensitive code may be
16 executed based on the dynamic permissions available at the demand point at
17 runtime. If the demand is not satisfied, an exception is thrown and the sensitive
18 code is not executed.

19 However, security gaps in the trusted components 112 can severely
20 compromise the security of the entire system 100. Therefore, it is important that
21 security gaps in trusted components are located and corrected before the trusted
22 components are deployed. Therefore, the security of the trusted components 112
23 in FIG. 1 has been reviewed by their developers prior to their deployment into the
24 managed execution environment 102.

25

1 In the illustrated implementation, a set of test permissions was specified
2 (e.g., by a developer) to characterize the level of trust attributed to an arbitrary
3 untrusted program. Based upon this characterized level of trust, an analysis tool
4 generated a call graph associated with the trusted components 112 to assist the
5 developer in identifying potentially non-secure code points within the trusted
6 components.

7 In another implementation, a developer has also applied one or more
8 queries to the call graph to filter out call paths that are ostensibly unrelated to a
9 specific security concern. For example, a developer can apply a query pertaining
10 to the ability to execute a FILE.DELETE method in one of the trusted
11 components. The analysis tool evaluates the query against the call graph, the test
12 permissions, and the security framework to identify a subset of call paths that may
13 be executed by an arbitrary unknown applet, which is characterized by the
14 specified test permissions, to satisfy the query. The developer can then evaluate
15 this identified subset of the call paths in the call graph to determine whether a
16 security concern exists. By generating this subset of call paths, the analysis tool
17 greatly decreases the effort required to identify non-secure methods and, therefore,
18 improves the overall security of the trusted components 112 executing in a
19 managed execution environment. In addition, the tool may also generate its own
20 queries and check the results automatically, flagging potentially unsafe code.

21 Access rights are represented in a managed execution environment using
22 runtime permissions, which include a collection of objects organized by classes.
23 Each such permission class details the rights to access specific resources. For
24 example, class UIPermission describes access to the user interface and has a
25 Boolean flag that controls access to the clipboard, and class FileIOPermission

1 describes read and write access to the file system, potentially for every possible
2 file path.

3 When a code assembly is loaded into the managed execution environment,
4 its access rights are determined by the runtime security policy, and the
5 corresponding static permissions are associated with every piece of code from that
6 assembly. These static permissions give an upper bound to the permissions that
7 the code can actually use. An exemplary default security policy may be to grant
8 most permissions to code written by the user, and restricted permissions to
9 downloaded code.

10 When examining access rights at runtime, before executing a sensitive
11 operation, the current dynamic permissions are compared to the permissions
12 requested for the operation. If the dynamic permissions do not include the
13 requested permission, a security exception is thrown. Otherwise, the operation
14 proceeds. As a first approximation, the dynamic permissions are computed as the
15 intersection of the static permissions for all methods on the current call stack. As
16 a refinement, any method may assert additional dynamic permissions for the
17 benefit of its callees, provided they are already included in the calling method's
18 static permissions

19 Dynamic permissions are not explicitly maintained by the managed
20 execution environment. Instead, they are extracted from the stack on-demand,
21 whenever a security check is performed. This exemplary approach, referred to as
22 "stack inspection", proceeds as follows:

23 A security check requests a check for a requested permission
24 (e.g., permission to write to a given file). For each frame on the
25

1 stack, starting from the most recent frame, the code pointer in the
2 frame is used to retrieve the static permissions for that code. If a
3 requested permission is not included in these static permissions, then
4 the inspection fails and a security exception is thrown. Otherwise, if
5 the frame being considered explicitly grants the requested
6 permission, the inspection immediately succeeds. Otherwise, the
7 next frame is considered.

8 In addition, some access rights may be examined at load-time. When
9 loading the code of a component that calls a method with a “link-demand” request,
10 the managed execution environment checks that the requested permission is
11 included in the static permissions of the caller. Otherwise, the link-demand fails
12 and a security exception is thrown.

13 In this context, a permission-sensitive call graph may be generated by
14 simulating the mechanics of the trusted component code within the context of a
15 given security framework. Thereafter, the resulting call graph may be analyzed
16 relative to a specific set of queries to identify possible security gaps.

17 FIG. 2 illustrates an exemplary analysis tool 200 for reviewing trusted
18 component code for execution in a test configuration of a managed execution
19 environment. In a typical scenario, a trusted component developer or tester
20 provides trusted component code 202 and a runtime security policy for unknown
21 code 204 as inputs to the tool 200. The trusted component code 202 includes the
22 program code and data for one or more trusted components that the developer
23 wishes to test within the security framework.

The runtime security policy 204 represents a hypothesis of the static permissions attributed to an unknown program (e.g., an applet) that may be executed within the test configuration using the provided trusted component code 202. Using this runtime security policy 204 and the analysis tool 200, the developer can review the trusted component code 202 for security gaps using a resulting call paths set 206. Alternatively, the runtime security policy 204 is obtained from the security framework 212 for input to the analysis tool.

In order to abstractly represent sets of permissions in one implementation, both in the input stage and during analysis, a symbolic representation for permissions is employed in combination with an associated symbolic implementation of the operations on permissions, such as Assert and Demand. The choice and precision of the symbolic representations can be controlled by the target permission scoping parameters 209 that may be input to the analysis. The target permission scoping parameters 209 specifies the subset of permissions that the developer or tester wishes to analyze. By specifying this subset, performance of the call graph generation and analysis may be improved.

It should also be understood that, if performance allows, a non-symbolic representation of permissions may be employed. For example, a permission set may be represented non-symbolically using a machine word if the number of permissions is no more than the width of the word.

In various implementations, therefore, the permissions may be represented in a variety of ways (as controlled by the target permissions scoping parameters 209) including without limitation:

(1) Permissions may be represented symbolically by two values:
NO_PERMISSIONS (representing any sets of permissions that

are statically granted to unknown code) and **SOME_PERMISSIONS** (representing any sets of permissions).

- (2) Permissions may be represented symbolically by an array of the two values, NO_PERMISSIONS and SOME_PERMISSIONS, indexed by a fixed collection of permission classes, thereby representing permissions for each of these classes independently.
- (3) Permissions may be represented explicitly in the same representation used in the security framework.
- (4) Some permissions may be represented symbolically while some permissions may be represented explicitly. This allows permission analysis when certain parameters of the permissions (e.g., the exact file name used in a specific permission to delete that file) cannot be determined during the analysis.
- (5) Permissions may be represented differently for each usage of permissions in the analysis. For example, a more precise symbolic representation may be represented in the second stage of analysis than in the first stage of analysis.

For a given choice of symbolic representation, corresponding symbolic operations are defined in order to approximate (i.e., typically a conservative approximation) the operations in the system. For example, when permissions are represented by the two values identified in item (1) above, an operation of adding a permission to “NO_PERMISSIONS (a) yields NO_PERMISSIONS, if the permission is statically granted to unknown code, and (b) yields SOME_PERMISSIONS otherwise. Likewise, performing a Demand of a

1 permission not included in the static permissions granted to unknown code always
2 fails on NO_PERMISSIONS and may or may not succeed on
3 SOME_PERMISSIONS.

4 In order to extract the symbolic representation of the permissions that are
5 used in an input of the analysis, and in particular, in the security policy and actions
6 of the known code, an auxiliary analysis may be employed. For example, a
7 security Demand in a piece of code usually takes as a parameter a permission that
8 is locally constructed or that is stored in a private field. Therefore, detailed
9 information on that permission may be obtained by a local-data-flow analysis.
10 Note that the auxiliary analysis may return a symbolic representation that
11 approximates the resulting permissions in complex cases.

12 The trusted component code 202 and the runtime security policy 204 are
13 received by a call graph generator 208, which generates a permission-sensitive call
14 graph. Let a “node” be represented by a pair(M,D), where D is a symbolic
15 permission set and M is either (a) a known method implementation in a trusted
16 component or (b) a token representing an unknown method implementation
17 belonging to some unknown component, representing arbitrary untrusted or
18 partially trusted code. The permission-sensitive call graph consists of a set of
19 nodes and a set of directed, labeled edges. Each edge extends from one node (the
20 symbolic caller) to another (the symbolic target). The label of an edge includes
21 (1) a call-site within the method implementation of the node and (2) a sequence of
22 security actions preceding the call-site within the method implementation. These
23 actions represent operations on permissions within the method, including
24 Demands and Asserts with their symbolic permission parameters.

1 Accordingly, a given piece of code that can be executed within different
2 dynamic security contexts may be represented as several nodes in the call graph,
3 each with a different symbolic set of dynamic permission and potentially different
4 outgoing edges.

5 The call graph generator 208 takes into account the particular semantics of
6 the underlying runtime security framework 212, including without limitation the
7 object model, the type system, the access modifiers, the resolution of virtual calls,
8 and the inheritance hierarchy. In this context, the call graph generator 208
9 analyzes the trusted component code 202 and the runtime security policy 204, as
10 they would apply to unknown, arbitrary code, to generate the call graph 210.

11 The resulting call graph 210 is stored in a computer-readable storage
12 medium (e.g., memory or a hard disk) and then input to a call graph analyzer 214.
13 In addition, one or more queries in a query set 216 are input to a call graph
14 analyzer 214. Some queries may be automatically or previously generated and
15 then checked by the call graph analyzer 214. Security gaps identified by the
16 automatic checking are output as security reports 218 on possible security
17 vulnerabilities or as identified call paths in a call paths set 206. Other queries may
18 be manually generated by the developer or tester. The call graph analyzer 214
19 outputs the call path set 206, which includes a subset of call paths that satisfy the
20 queries. Some queries can also involve auxiliary analysis of the code and any
21 information provided by the call path set 206.

22 FIG. 3 illustrates exemplary operations 300 for generating a permission-
23 sensitive call graph. For a given input configuration, a call graph is deemed
24 “correct” when, (a) for any configuration extended with additional component
25 code that is accepted by the runtime security policy characterizing the unknown

1 code and (b) for every runtime call from one piece of code to another, there is (1) a
2 corresponding edge between two nodes associated with those two pieces of code
3 and (2) the runtime dynamic permissions for these pieces of code correspond to
4 the symbolic dynamic permissions for their nodes.

5 A hierarchy operation 302 receives the trusted component code and
6 generates a class hierarchy that contains the classes of the trusted components plus
7 symbolic classes that represent any class that may be defined in unknown
8 components. For each class in an unknown component that can be added to the
9 system according to the security policy, there exists a corresponding symbolic
10 class in the completed class hierarchy. Hence, a single symbolic class in the
11 analysis can represent many possible classes in unknown component.

12 In one implementation, the completed class hierarchy may be generated by
13 simulating the mechanisms of the system, including its type discipline and its
14 security policy, and may depend in particular on the class definitions in the trusted
15 components and on the runtime security policy characterizing the unknown code.
16 This simulation may also depend upon rules of class and interface inheritance,
17 access modifiers (e.g., public, private, virtual, sealed, etc.), and on specific
18 declarative security attributes (e.g., parametric InheritDemands).

19 In languages with virtual calls, the generation of the call graph may also
20 involve the symbolic representation of runtime values. For example, runtime
21 values may be represented as sets of dynamic classes in the completed class
22 hierarchy.

23 A generation operation 304 generates initial constraints to determine an
24 initial value of a symbolic value V , which represents all values that may be
25 obtained by any unknown code at runtime (e.g., as the result of calls to known

1 method implementations, as the parameters of callbacks from known to unknown
2 code, through shared data structures, etc.). The symbolic value V can be updated
3 during the analysis as new dataflow to unknown code is revealed.

4 An identification operation 306 uses V to identify trusted code methods that
5 are directly callable by unknown code. In one implementation, for example, these
6 identified methods may include public methods of trusted classes, protected
7 methods of trusted classes (if the class is not sealed), and methods in an accessible
8 assembly. In addition, if the unknown code has obtained a value compatible with
9 the instance type, the method may be identified unless the method has a link-
10 demand on a permission that is not included in the static permission of the
11 unknown code.

12 For each such identified method, a generation operation 308 adds a node
13 with appropriate constraints to the call graph. Constraint generation may identify
14 additional trusted code methods and result in insertion of additional nodes in the
15 call graph and generation of additional constraints. In one implementation,
16 constraint generation relies on the generation of an intra-method control-flow
17 graph, which depends on branching instructions, exception handling, etc. The
18 intra-method control-flow graph propagates information on the (usually empty)
19 series of local security actions that have been performed since the method entry
20 point. To obtain an efficient implementation of the graph construction, tables may
21 be used to cache the result of many computations, such as the symbolic evaluation
22 of actions on permissions, the accessibility checks, and the resolution of method
23 references to method implementations.

24 In one implementation, the identification of trusted code methods to be
25 included in the call graph is accomplished by generation and resolution of

1 constraints. Generally, constraints simulate the dataflow of symbolic values
2 manipulated by trusted code and may also simulate the dataflow of symbolic
3 values between trusted code and unknown code.

4 Constraint resolution operation 310 propagates the symbolic values, which
5 may result in insertion of additional nodes in the call graph and generation of
6 additional constraints. When a call to a native method, which is implemented
7 outside the managed execution environment, is performed, and in other
8 circumstances when a call cannot be symbolically traced during analysis, a
9 symbolic representation of any values that have the static return type of the
10 method is assigned to the result of the call. For example, if values are represented
11 as sets of classes, the return variable is assigned the set of all subclasses of the
12 static return class in the completed class hierarchy.

13 Direct calls to a given method reference R on a given object of type T
14 (either present in the code or generated by solving virtual-call constraints), are
15 solved in one implementation by:

- 16 • Computing the method implementation M associated with R and T ,
17 according to the method resolution rules of the system;
- 18 • Computing the (symbolic representation of the) dynamic
19 permissions D' for that method implementation as the intersection of
20 the dynamic permissions D of the caller for that call site and of the
21 static permissions S_M of the callee;
- 22 • Unless a node(M, D') already exists, create this node and recursively
23 generate the constraints for that node;

24 Record an edge from the call-site to that node; and

25

1 Generate constraints for each parameter of the call and for its
2 returned result (if any), depending on the type signature of the
3 method reference.

4 Virtual calls are common cases of inter-method control flow. In one
5 implementation, virtual call constraints are solved as follows:

- 6 • For each dynamic class of the input variable associated with the
7 object used for the virtual call,
 - 8 ▪ If the class is a trusted class, then perform a direct call to the
9 method reference on that class type.
 - 10 ▪ If the class is a symbolic class representing some unknown
11 code,
 - 12 ➢ If the symbolic class can inherit a method implementation
13 from a trusted parent class or interface, then perform a direct call
14 to that method implementation, as detailed above; and
 - 15 ➢ If the symbolic class can provide a method implementation
16 for that signature, then record an edge from the call site to
17 unknown code and, for each parameter and for the returned result
18 (if any), generate a constraint that safely approximates a dataflow
19 with unknown code (i.e., any value passed as a parameter
20 becomes available to unknown code and is added to the symbolic
21 value V ; and any (type-safe) value can be returned by the
22 unknown code).

23 When an instruction triggers a dynamic action on permissions (e.g., by
24 performing a virtual call to the Demand method), a special conditional constraint
25 is generated from the local parameters of the call. This constraint is solved by

1 considering the values that flow to the variable representing the permission used to
2 perform the action. For example, in the case of a Demand P , it can be determined
3 whether the value that flows to P may be included in the dynamic permissions of
4 the node being analyzed. If so, the analysis is resumed on the piece of code
5 guarded by the Demand. Otherwise, the Demand would fail at runtime and the
6 analysis need node consider the code guarded by the Demand.

7 In the case of an Assert P , the value that flows to P is used to compute an
8 updated set of dynamic permissions D' as the union of D and P intersected with
9 the static permission for that code, S . The analysis is then resumed on the piece of
10 code guarded by the Assert with dynamic permissions D' .

11 An alternative implementation for Asserts and Demands assumes that a
12 single permission value flows to that variable P and validates that assumption at
13 the end of the analysis. Pragmatically, this assumption is almost always true in
14 trusted libraries, thereby enabling a more precise symbolic simulation of the
15 security action. Independently, the resolution of security actions can
16 advantageously be deferred, so as to generate first a call graph that does not
17 depend on the security policy and can be completed into different call graphs for
18 different security policies.

19 Actions on permissions that are programmed using declarative security
20 attributes may also be handled as described above.

21 When an object is created, the symbolic result represents only values with
22 the dynamic class of the object constructor. When the value of an expression is
23 stored into a variable (such as a local variable, an entry on the stack, a parameter
24 of a method, or a method result), a constraint is generated that states that all the
25

1 symbolic values taken by the expression are included in the symbolic values taken
2 by the variable.

3 A decision operation 312 determines whether additional propagation is
4 possible, based on whether all symbolic constraints are satisfied. If so, the next
5 iteration is performed starting at the identification operation 306. Otherwise, a
6 storage operation 314 stores the resulting call graph in a computer-readable
7 medium. The resulting call graph is a conservative representation of all potential
8 call paths initiated by the arbitrary unknown code that is represented by the input
9 runtime security policy.

10 The resulting call graph provides an informative context to assist the
11 developer or tester in reviewing the code for security risks. Some exemplary
12 structural properties of the resulting call graph are highlighted below:

- 13 (1) A call graph contains a set of security-sensitive methods that
14 may be reached in the presence of unknown code, as specified
15 by the input runtime security policy of the unknown code.
- 16 (2) The source for a given piece of code can be annotated with the
17 static permissions and the (potentially multiple) dynamic
18 permissions for that piece of code in the context of the global
19 security framework used to generate the graph. This information
20 is particularly helpful when these dynamic permissions are
21 strictly lower than the static permissions or when they may take
22 multiple values.
- 23 (3) Each call site can be annotated with the actual nodes (e.g.
24 method implementations and associated dynamic permissions)
25 that can effectively be called from that call site at runtime.

(4) When a piece of trusted code can call some unknown code (e.g., as a result of a virtual call), the call site can be annotated with a symbolic description of the unknown callee, such as “any implementation of method M in a subclass of class C”. This information is useful to detect the escape of sensitive values passed as parameters to unknown code.

After a call graph is generated, a developer or tester can display the symbolic representations of any values that are accessible to unknown code. In addition, the developer or tester can highlight reachable method implementations that perform sensitive actions that operate on permissions, that may call unknown code, or that can be executed with (symbolically) different sets of dynamic permissions.

In addition, for each sensitive action, the call graph provides a collection of paths that lead from unknown code to that particular action. For example, a system call that deletes a file is a sensitive action and should be reachable from unknown code only in specific circumstances with adequate security checks that can be assessed for all call paths from unknown code to that system call.

In this context, a permission-sensitive control-flow analysis can provide valuable information to a developer or tester of trusted code components to confirm compliance with a security framework.

One aspect of this analysis involve the input of one or more queries (whether manual or automatic) to flag potential security vulnerabilities and to narrow the number of call paths that are included in the resulting call paths set for review by the developer or tester. Queries may be expressed as structural properties on call paths, such as “All paths from unknown code to any sensitive

1 action in a given set" or "All paths with a Demand followed by an Assert for a
2 given permission class". The queries are used to evaluate the call graph to yield a
3 set of paths that can be reported to the developer or tester using an exemplar for
4 each class of paths that satisfies the structural properties by appealing to an
5 adequate definition of equivalence between paths. For example, two paths are
6 equivalent if they contain the same interleaving of security actions, irrespective of
7 intermediate calls, and the exemplar is a path with the least number of
8 intermediate calls. Once a particular path has been identified to satisfy a query,
9 the path in the call graph can be displayed as a symbolic runtime stack
10 representing a series of nested calls interleaved with permission operations.

11 A call graph analyzer can also perform a wide variety of security checks on
12 trusted code components. Exemplary checks are discussed below.

13 In one implementation, use of permissions, such as in Demand and Assert
14 instructions, is checked. Each dynamic operation on permissions should have a
15 clear purpose. As such, dynamic operations on permissions that are defined in the
16 call graph and that have an unclear purpose may reveal a security problem in the
17 system. Therefore, using queries for each dynamic permission operation (e.g.,
18 Demand or Assert) can check whether the security programming is adequate.

19 One rule that can be applied is that every permission Demand should be
20 necessary. For each "Demand P " followed by a piece of code, the analysis
21 module checks to determine whether the permission Demand is necessary by:

22 (1) Verifying that the Demand may fail at runtime, thereby justifying its
23 runtime cost. Such a check may be achieved by identifying a node in the call
24 graph for the method implementation with that demand having dynamic
25 permissions D that do not necessarily include P . If no such node is identified, the

1 Demand may be flagged as redundant in the test configuration, which may warrant
2 removing the Demand from the trusted component code or reviewing the
3 demanded permission P (wherein P is referred to as a parameter permission of the
4 Demand).

5 (2) Recursively computing the method implementations that are (a)
6 reachable from that protected piece of code and (b) only reached at nodes with
7 dynamic permissions D that include P , irrespective of the caller. These method
8 implementations can be used to document the security need for coding the
9 Demand. In case there is no such method implementation, the Demand can be
10 flagged as potentially erroneous in that test configuration. Similarly, relying on
11 classification of sensitive actions in the code, the call graph can be used to
12 associate with each Demand a set of sensitive actions that are reachable from the
13 piece of code protected by the Demand.

14 Another rule that can be applied is that every permission Assert should be
15 necessary and should have a minimal scope, thereby complying with the general
16 “least privilege principle” of system security. For each “Assert P_a ” followed by a
17 piece of code, the analysis module checks to determine whether the permission
18 Assert is necessary and has minimal scope by:

19 (1) Verifying that the Assert may be necessary at runtime, therefore
20 justifying its runtime cost. Such a check may be achieved by identifying a node
21 for the method implementation associated with the Assert and having dynamic
22 permissions D that do not necessarily include P_a . Otherwise, the Assert can be
23 flagged as redundant in the test configuration, which may warrant removing the
24 Assert from the trusted component code.

25

1 (2) (a) Recursively computing the nodes that are (i) reachable from that
2 privileged piece of code and (ii) only reached at nodes with dynamic permissions
3 D that may include P_a and (b) collecting all reachable permission Demands in
4 these nodes with permissions P_d included in P_a , irrespective of the caller. These
5 method implementations can be used to document the security need for coding the
6 Assert. In case there are no such Demands, the Assert can be flagged as
7 unnecessary. In case there exists a permission P'_a weaker than P_a that contains all
8 demanded permissions P_d , the asserted permission P_a can be flagged as
9 unnecessarily high. In case parts of the code protected by the Assert do not reach
10 any associated Demand, the scope of the Assert can be flagged as unnecessarily
11 large in the test configuration. (The asserted permission P_a is also referred to as a
12 parameter permission of the Assert.) In one implementation, the use of matching
13 “RevertAssert” security actions can be used to cancel the effect of the Assert
14 immediately after the last call to a method that reached an associated Demand.

15 In combination, the two types of queries described above can be used to
16 suggest moving Asserts and Demands within the trusted component code in order
17 to minimize their runtime cost while providing an adequate level of protection. In
18 addition, for every public method that can reach a Demand that may fail for a
19 given permission, the annotations associated with that node may be checked to
20 verify that the Demand in the method with that particular permission may succeed.

21 As an example, a piece of code may Demand a permission before calling a
22 “protected” method, whereas the same method can be called directly by a class in
23 unknown code that inherits from the implementation class. A query can detect and
24 report this situation, with two likely explanations: (1) the Demand is useless; or
25 (2) the implementation class should be “sealed”.

1 Yet another rule that can be applied is that security check should be
2 uniform: for a given protected sensitive action, and for every control path to the
3 sensitive action, the series of security actions should implement the same
4 (implicit) security specification. More generally, permission classes define a data
5 structure, rather than a security policy, and the information collected by the
6 analysis should provide a synthetic view of the usage of these permission classes
7 to protect sensitive operations.

8 Therefore, using the graph for a given set of sensitive actions, the nodes of
9 the graph may be partitioned into “ordinary nodes” and a “protected subsystem”.
10 Furthermore, every edge can be checked to ensure that each edge from an ordinary
11 node to the protected subsystem corresponds to a piece of code that implements
12 adequate security checks for these sensitive actions. Once identified, the
13 protective “boundary” and the contents of the protected subsystem provide a clear
14 summary of access control for these sensitive actions. Using the call graph, the
15 partitioning can be largely automated, for example, by defining the “protected
16 subsystem” as the set of nodes with dynamic permissions D containing the
17 particular permission intended to protect these sensitive actions or defining the
18 protective boundary of the protected subsystem as the set of nodes that meet a path
19 query on the graph.

20 Yet another rule that may be applied involves a comparison between the
21 results of identical queries on different call graphs, generated from different input
22 configurations. This approach is particularly useful when comparing similar
23 configurations, such as comparing a “reference” configuration that has been
24 extensively analyzed and tested to an “updated” version of that configuration.

1 For example, a developer or tester may check and carefully review any new
2 paths leading to sensitive operations in the call graph for the “updated”
3 configuration, under the assumptions that all previously-existing paths have
4 already been reviewed. Typical applications of this comparative analysis involve
5 an updated configuration with:

- 6 • A change in the static permissions attributed to some components
- 7 • The addition of a new trusted component to a much-larger collection
8 of components, including standard libraries.
- 9 • The update of a few method implementations in a component, for
10 example, to fix a security error or to extend their functionality.

11 Yet another rule that can be applied involves the checking of link-demand
12 usage. For performance reasons, declarative “link-demands” may be used in place
13 of Demands, with the same security intent. It can be verified that link-demands
14 and Demands have the same impact on control flow. Furthermore, Demands that
15 can be safely replaced with link-demands or discarded can be identified. If, for
16 every node of the call graph associated with a method implementation, the
17 dynamic permissions D contain the demanded permission P , then the operations
18 link-demand P and Demand P are equivalent. Otherwise, any path from unknown
19 code to the node with dynamic permissions that lack P can pass the link-demand
20 but would cause a Demand to fail. Such paths may be flagged for manual review.
21 In addition, a call graph can be used to visit all (potential) dynamic callers of a
22 particular method with a dynamic Demand. This usage of the call graph
23 facilitates manual consideration of whether replacing the Demand with a link-
24 demand to a permission specific to these callers is warranted.

1 Queries on the call graph can also be used to determine whether a code
2 transformation is correct in a given test configuration. The responses to the
3 queries may be passed to some other part of the system, such as an optimizing
4 compiler, either at runtime or by annotating the code in advance. For example:

- 5 • A method call may be inlined by replacing the call to that method by
6 the code that implements the method. This transformation is correct
7 if the dynamic permissions for the inlined code are not affected. In
8 most cases, a series of simple checks on the code and its
9 corresponding nodes in the call graph suffice to ensure that the
10 transformation is correct.
- 11 • Similarly, the correctness of a tail call elimination can be statically
12 guaranteed by simple checks on the code and the call graph.
- 13 • More specifically, queries detailed above may be run on the call
14 graph to determine whether the relocation of security actions, such
15 as Assert and Demand, or even their elimination, is correct for a
16 given configuration. For example, if a Demand always succeeds, it
17 can safely be eliminated from the code as it is loaded into the
18 system.

19 A permission-sensitive call graph can also be used to explore the
20 consequences of changing the static permissions attached to known components,
21 which can be controlled using “Declarative Requests” attached to these
22 components. For example, before removing a particular permission P from the
23 static permissions of a given component, a list may be generated of all nodes with
24 code from that component that is reachable with dynamic permissions that may
25 contain P . From this list, a report of all Demands that are reachable from these

1 nodes and that may fail as a result of the change (i.e., the removal of the
2 permission P) may be computed. Conversely, the call graph may be used to
3 identify such permissions P for a given configuration. In the special case of a
4 closed program (i.e., one with no unknown components), a call graph may be used
5 to determine which Demands in the closed program may fail at runtime.

6 The permission-sensitive call graph can also be used to check several
7 specific properties on permissions. Various optimizations may be apparent using
8 the call graph and some additional local checks for each reachable security action:

- 9 • Some arguments of permissions, such as file name expressions for
10 System.FileIOPermissions, must be appropriately normalized.
- 11 • For efficiency reasons, frequently used permission objects should be
12 allocated once and for all. Similarly, security actions should
13 preferably be moved outside execution loops.
- 14 • Security actions, such as Asserts and Demands, can be programmed
15 using either dynamic method calls or declarative security attributes.
16 The trade-off between the two choices is expressiveness versus
17 performance. The call graph provides useful information for
18 deciding which choice is best.

19 The permission-sensitive call graph can also be used to evaluate security
20 risks introduced by changes to the security infrastructure. For example, if new
21 trusted libraries are made accessible to untrusted code or if the security model is
22 changed, the call graph may be used to determine whether any new security
23 vulnerabilities are introduced by the changes.

24 The exemplary hardware and operating environment of FIG. 4 for
25 implementing the invention includes a general purpose computing device in the

1 form of a computer 20, including a processing unit 21, a system memory 22, and a
2 system bus 23 that operatively couples various system components include the
3 system memory to the processing unit 21. There may be only one or there may be
4 more than one processing unit 21, such that the processor of computer 20
5 comprises a single central-processing unit (CPU), or a plurality of processing
6 units, commonly referred to as a parallel processing environment. The computer
7 20 may be a conventional computer, a distributed computer, or any other type of
8 computer; the invention is not so limited.

9 The system bus 23 may be any of several types of bus structures including a
10 memory bus or memory controller, a peripheral bus, a switched fabric, point-to-
11 point connections, and a local bus using any of a variety of bus architectures. The
12 system memory may also be referred to as simply the memory, and includes read
13 only memory (ROM) 24 and random access memory (RAM) 25. A basic
14 input/output system (BIOS) 26, containing the basic routines that help to transfer
15 information between elements within the computer 20, such as during start-up, is
16 stored in ROM 24. The computer 20 further includes a hard disk drive 27 for
17 reading from and writing to a hard disk, not shown, a magnetic disk drive 28 for
18 reading from or writing to a removable magnetic disk 29, and an optical disk drive
19 30 for reading from or writing to a removable optical disk 31 such as a CD ROM
20 or other optical media.

21 The hard disk drive 27, magnetic disk drive 28, and optical disk drive 30
22 are connected to the system bus 23 by a hard disk drive interface 32, a magnetic
23 disk drive interface 33, and an optical disk drive interface 34, respectively. The
24 drives and their associated computer-readable media provide nonvolatile storage
25 of computer-readable instructions, data structures, program modules and other

1 data for the computer 20. It should be appreciated by those skilled in the art that
2 any type of computer-readable media which can store data that is accessible by a
3 computer, such as magnetic cassettes, flash memory cards, digital video disks,
4 Bernoulli cartridges, random access memories (RAMs), read only memories
5 (ROMs), and the like, may be used in the exemplary operating environment.

6 A number of program modules may be stored on the hard disk, magnetic
7 disk 29, optical disk 31, ROM 24, or RAM 25, including an operating system 35,
8 one or more application programs 36, other program modules 37, and program
9 data 38. A user may enter commands and information into the personal computer
10 through input devices such as a keyboard 40 and pointing device 42. Other
11 input devices (not shown) may include a microphone, joystick, game pad, satellite
12 dish, scanner, or the like. These and other input devices are often connected to the
13 processing unit 21 through a serial port interface 46 that is coupled to the system
14 bus, but may be connected by other interfaces, such as a parallel port, game port,
15 or a universal serial bus (USB). A monitor 47 or other type of display device is
16 also connected to the system bus 23 via an interface, such as a video adapter 48.
17 In addition to the monitor, computers typically include other peripheral output
18 devices (not shown), such as speakers and printers.

19 The computer 20 may operate in a networked environment using logical
20 connections to one or more remote computers, such as remote computer 49. These
21 logical connections are achieved by a communication device coupled to or a part
22 of the computer 20; the invention is not limited to a particular type of
23 communications device. The remote computer 49 may be another computer, a
24 server, a router, a network PC, a client, a peer device or other common network
25 node, and typically includes many or all of the elements described above relative

1 to the computer 20, although only a memory storage device 50 has been illustrated
2 in FIG. 4. The logical connections depicted in FIG. 4 include a local-area network
3 (LAN) 51 and a wide-area network (WAN) 52. Such networking environments
4 are commonplace in office networks, enterprise-wide computer networks, intranets
5 and the Internet, which are all types of networks.

6 When used in a LAN-networking environment, the computer 20 is
7 connected to the local network 51 through a network interface or adapter 53,
8 which is one type of communications device. When used in a WAN-networking
9 environment, the computer 20 typically includes a modem 54, a network adapter, a
10 type of communications device, or any other type of communications device for
11 establishing communications over the wide area network 52. The modem 54,
12 which may be internal or external, is connected to the system bus 23 via the serial
13 port interface 46. In a networked environment, program modules depicted relative
14 to the personal computer 20, or portions thereof, may be stored in the remote
15 memory storage device. It is appreciated that the network connections shown are
16 exemplary and other means of and communications devices for establishing a
17 communications link between the computers may be used.

18 In an exemplary implementation, a call graph generator, a call graph
19 analyzer, a security framework, and other modules may be incorporated as part of
20 the operating system 35, application programs 36, or other program modules 37.
21 A call graph, a call path set, an input permission set, one or more queries, and
22 other data may be stored as program data 38.

23 The embodiments of the invention described herein are implemented as
24 logical steps in one or more computer systems. The logical operations of the
25 present invention are implemented (1) as a sequence of processor-implemented

1 steps executing in one or more computer systems and (2) as interconnected
2 machine modules within one or more computer systems. The implementation is a
3 matter of choice, dependent on the performance requirements of the computer
4 system implementing the invention. Accordingly, the logical operations making
5 up the embodiments of the invention described herein are referred to variously as
6 operations, steps, objects, or modules.

7 The above specification, examples and data provide a complete description
8 of the structure and use of exemplary embodiments of the invention. Since many
9 embodiments of the invention can be made without departing from the spirit and
10 scope of the invention, the invention resides in the claims hereinafter appended.

11
12
13
14
15
16
17
18
19
20
21
22
23
24
25